National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file - Votre référence

Our file - Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

Parallel Domain Decomposition of a Heat Equation

BY

Oleg Verevka

©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

DEPARTMENT OF MATHEMATICS

Edmonton, Alberta
Fall 1993

National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-88365-0

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Oleg Verevka

TITLE OF THESIS: Parallel Domain Decomposition of a Heat Equation

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1993

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

(Signed) Oleg Verevka

Oleg Verevka
45^a , Ivan Franko str, ap.18
Odessa , 270049
Ukraine

Date: October 8, 93

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Parallel Domain Decomposition of a Heat Equation** submitted by Oleg Verevka in partial fulfillment of the requirements for the degree of Master of Science.

. . . . . .
Dr. Yanping Lin (Supervisor)

. . . . . .
Dr. Jonathan Schaeffer (External)

. . . . . .
Dr. Yau Shu Wong (Chair)

Date: Sept. 21, 93

Abstract

Domain decomposition methods have proved to be an efficient approach for parallel processing of partial differential equations on parallel architectures. The method is well developed for elliptic equations, while it is still one of the active research areas for the other types of mathematical models.

This work studies applications of geometric domain decomposition for a heat equation in cylindrical coordinates. The explicit/implicit algorithm, initially proposed by Li, Lin and Wong, is implemented on a MYRIAS SPS-2 parallel computer. The main objective of the research is to achieve a high parallel performance of the method by reduction of communication and synchronization barriers. The thesis introduces various asynchronous modifications of the original technique and investigates their stability and accuracy of approximation.

Contents

1 Introduction 1

1.1 Formulation of a heat conduction problem in cylindrical coordinates . 1

1.2 Sequential numerical methods 3

1.3 Finite difference domain decomposition 4

1.4 A speedup ratio as a measure of algorithm efficiency 6

2 Direct Implementation of Domain Decomposition Algorithm 8

2.1 Algorithm pseudocode, data structures and shared memory model . . 8

2.2 Dependance of the speedup on a problem size 15

2.3 A geometry of domain decomposition. 16

3 Asynchronous Domain Decomposition Algorithm 20

3.1 Asynchronous chaotic algorithm 20

3.2 Adaptive asynchronous algorithm 23

3.3 Stability of asynchronous adaptive algorithm 26

4 Domain Decomposition with Reduced Communication Scheme 30

Bibliography 36

Appendix 37

List of Figures

1 The sub-domain data structures 9

2 Influence of page alignment of shared memory 14

3 Speedup graph of direct implementations 19

4 A computation halo of combined correction operator 25

5 Speedup of a chaotic algorithm with corrections 27

6 A typical computation log-file 29

7 Reduced communication verses other methods 32

List of Tables

2.1 Dependance of the speedup and approximation errors on a problem size 15

2.2 Synchronization overhead of equally partitioned domains 17

2.3 Synchronization overhead of a "heavy border" distribution 18

2.4 Synchronization overhead of a "mixed distribution" 18

3.5 Convergence of the asynchronous method 22

3.6 Stability of adaptive asynchronous scheme 28

4.7 Stability of the reduced communication scheme 31

4.8 Dependence of stability on a number of sub-domains. 32

A.9 Performance of the algorithm verses domain size 39

A.10 Statistics for the equal distribution algorithm 39

A.11 Statistics for the "heavy border" distribution algorithm 39

A.12 Statistics for the mixed distribution algorithm 40

A.13 Statistics for asynchronous algorithm 40

A.14 Statistics for adaptive asynchronous algorithm ($K = 8$) 40

A.15 Statistics for the algorithm with reduced communication scheme ($K = 2$) 41

A.16 Statistics for the algorithm with reduced communication scheme ($K = 4$) 41

A.17 Statistics for the algorithm with reduced communication scheme ($K = 6$) 41

Chapter 1

Introduction

1.1 Formulation of a heat conduction problem in cylindrical coordinates

A mathematical formulation of many problems in physics and engineering leads to a partial differential equation. In this paper we will study a classical parabolic initial value problem — a heat conduction problem on a disc. It can be defined in the following way.

Definition 1 *Suppose a function $\phi(r)$ represents a heat distribution on a disc of radius $r = 1$ at initial time $t_0 = 0$. Find function $u(r,t)$ that models the heat conduction process between time $t_0 = 0$ and $t_1 = 1$, if the temperature on the border of the disc is $u(1,t) = 0$ for any time $0 \leq t \leq 1$.*

This physical problem is equivalent to the homogeneous initial value problem (1.1):

$$\frac{\partial u}{\partial t} - \frac{1}{r}(ru_r)_r = 0 \tag{1.1}$$

$$0 < r < 1, \qquad 0 \leq t \leq 1;$$

$$u_r(0,t) = 0, u(1,t) = 0, \qquad 0 \leq t \geq 1;$$

$$u(r,0) = \phi(r), \qquad 0 < r < 1.$$

The above formulation assumes that the system is conserved and no additional sources of energy are present, otherwise parabolic initial value problem (1.1) is modified to:

$$\frac{\partial u}{\partial t} - \frac{1}{r}(ru_r)_r = F(r,t). \tag{1.2}$$

In order to test different numerical schemes all the computational experiments are going to be conducted for a non-homogeneous parabolic problem (1.2), while in all the formulas we will use homogeneous equation (1.1). The following two functions will be used as the right part $F(r,t)$ in (1.2):

- A function

$$F(r,t) = e^{-t}(cos(\frac{\pi}{2}r)((\frac{\pi}{2})^2 - 1) + \frac{\pi}{2r}sin(\frac{\pi}{2}r)) \tag{1.3}$$

that corresponds to a smooth exact solution:

$$u(r,t) = e^{-t}cos(\frac{\pi}{2}r). \tag{1.4}$$

This right part of the non-homogeneous problem was used for all the performance evaluation experiments.

- A function

$$F(r,t) = (4 - \frac{1-r^2}{\omega+t})\frac{\omega}{\omega+t} \tag{1.5}$$

that generates a solution:

$$u(r,t) = \frac{\omega}{\omega+t}(1-r^2). \tag{1.6}$$

The function (1.6) has a singularity in t on a line $t = -\omega$. Thus for very small ω in the close neighborhood of $t = 0$ the $\partial^2 u/\partial t^2$ is very big.

Notice, that (1.2) with (1.3) and (1.5) represent two artificial initial value problems and are chosen to demonstrate the accuracy of approximation and stability of numerical schemes for exact solutions with different properties.

1.2 Sequential numerical methods

In most cases, exact solutions of partial differential equations are impossible to find theoretically, but they can be approximated in practice by various numerical methods. Some of the most popular traditional techniques involve finite difference approximation.

Let us take N points of the disc radius $r_i = hi, i = 0, 1 \dots N-1$ with a space-step $h = 1/N$. Assume that a time interval $[t_0, t_1]$ is partitioned on M time-levels $t^n = \tau n, n = 1, 2, \dots M$, where $\tau = 1/M$ is a time-step. A numerical solution U_i^n of the heat equation will be computed for every time-level n at points r_i of the disc radius. Using the truncated Taylor expansion, derivatives $\frac{1}{r}(ru_r)_r$ and $\partial_t u$ can be approximated by the values of the solution at point (i, n) and it's neighbors. Mathematical formulas built in this manner are called **finite difference approximations**. Depending on the form of **a halo** — points in the mesh, whose values are used to approximate a solution at (r_i, t^n) — we can classify finite differences as **explicit forward difference:**

$$\frac{u_i^n - u_i^{n-1}}{\tau} = \frac{(r_i + h/2)(u_{i+1}^{n-1} - u_i^{n-1}) - (r_i - h/2)(u_i^{n-1} - u_{i-1}^{n-1})}{r_i h^2} \tag{1.7}$$

and **implicit backward difference:**

$$\frac{u_i^n - u_i^{n-1}}{\tau} = \frac{(r_i + h/2)(u_{i+1}^n - u_i^n) - (r_i - h/2)(u_i^n - u_{i-1}^n)}{r_i h^2}. \tag{1.8}$$

Any numerical algorithm can be called *computationally useful* if it is convergent, consistent and stable. In this paper our interest will be concentrated on stability of proposed computational schemes.

Definition 2 *Suppose that a numerical algorithm approximates the exact solution of the homogeneous problem (1.1) by a numerical solution U_j^n. If the following inequality holds:*

$$\max_n ||U_j^n|| < C||U_j^0|| \tag{1.9}$$

where C is a constant and $||\times||$ is one of the norms, then an algorithm is called **stable.**

In other words, a numerical algorithm is stable, if the maximum of computational errors is bounded by the input errors.

Presented methods (1.7) and (1.8) have the same degree of accuracy $O(h^2 + \tau)$, but represent two different schemes of computation.

The implicit backward difference method solves a system of linear equations for each time level. It is rather expensive. The main advantage of this scheme is its unconditional stability, i.e. there are no restrictions on a space-step h and time-step τ.

The explicit forward difference method uses an approximate relationship between the current time-level t_i and the next one t_{i+1} and does not require us to solve any linear equations. This method can be trivially parallelized, as computations of every point are independent from its neighbors on the same time-level, and rely only on the points of a previous level. The main disadvantage of the explicit method is its conditional stability. This method is stable only if:

$$\tau \leq \frac{h^2}{2}. \tag{1.10}$$

In comparison with the implicit method, this condition will force us to use more time-levels, than is necessary for a given accuracy.

1.3 Finite difference domain decomposition

With the rapid development of supercomputers, new numerical algorithms must be found. Domain decomposition technique is one of the most promising and actively studied areas in numerical analysis for parallel architectures. The main idea of this procedure is to divide a problem on a set of related subproblems of a smaller size and to solve them in a parallel manner. Although this method is already well developed for elliptic problems, it is still a research field for other types of equations.

We will study a geometric domain decomposition.

Definition 3 Geometric domain decomposition *is a process of breaking down a calculation into sections, which correspond to some physical sub-division of the system being modelled.*

In 1991 Du, Dawson & Dupont published a paper [1] in which a domain decomposition approach for a heat equation in Cartesian coordinates was studied. They proposed to use explicit forward finite difference method on the interfaces between sub-domains, while solving implicit backward difference linear equations for interior points in a parallel manner.

An application of the same technique for a heat equation in cylindrical coordinates is given in a recent publication of Li, Lin and Wong [2].

For a function $f(r,t)$ define $f_i^n = f(r_i, t^n)$ and

$$\partial_{t,\tau} f^n = \frac{f^n - f^{n-1}}{\tau}, \tag{1.11}$$

$$\partial^2_{r,h} f(r) = \frac{(r+h/2)(f(r+h)-f(r)) - (r-h/2)(f(r)-f(r-h))}{h^2}, \tag{1.12}$$

$$\partial_r f_0 = \frac{-3f_0 + 4f_1 - f_2}{2h}. \tag{1.13}$$

Assume, for simplicity, that a domain is decomposed into two sub-domains $0, 1, \ldots, d-1$ and $d+1, d+2 \ldots, N$, where d is an interface point. The numerical domain decomposition algorithm is defined in the following way:

- Interior points of both sub-domains $i = 1, 2, \ldots, d-1, d+1, \ldots, N-1$ will be computed using implicit method:

$$\partial_{t,\tau} U_i^n - \frac{1}{r_i} \partial^2_{r,h} U_i^n = 0. \tag{1.14}$$

- Suppose, the coarse grid space-step H is a multiple of h and satisfies the stability condition $\tau \leq H^2/2$. The interface point d can be solved by explicit forward difference formula:

$$\partial_{t,\tau} U_d^n - \frac{1}{r_i} \partial^2_{r,H} U_d^{n-1} = 0. \tag{1.15}$$

- In order to satisfy boundary conditions we will put:

$$U_N^n = 0, \quad n = 0, 1, \ldots, M, \tag{1.16}$$

and compute:

$$\partial_r U_0^n = 0, \quad n = 1, 2, \ldots, M. \tag{1.17}$$

- Initial conditions correspond to:

$$U_i^0 = \phi_i, \quad i = 0, 1, \ldots, N. \tag{1.18}$$

The following theorem, presented in [2], proves the stability of the described above method:

Theorem 1 *Assume, that U_i^n is a solution of (1.14)-(1.18) and $|\partial^2 u/\partial^2 t|$ and $|(ru_{rrr})_r|$ are bounded by a positive constant C_0 on $[0,1] \times [0,1]$. Then for $\tau = \Delta t \leq H^2/2$, the following error estimate holds:*

$$\max_{i,n} | u(x_i, t^n) - U_i^n | \leq C(C_0)(h^2 + H^3 + \tau). \tag{1.19}$$

1.4 A speedup ratio as a measure of algorithm efficiency

The main goal of parallel computing is, using multiprocessor architectures, to solve traditional problems significantly faster than can be done by a conventional sequential computer. The efficiency of any parallel algorithm and its implementation are usually judged by a speedup ratio.

Definition 4 [1] *If τ_1 is the time taken to run a program on one processor, and τ_p the time taken to run it on P processors, then the ratio of two program execution times:*

$$S = \frac{\tau_1}{\tau_p}$$

[1] This and many other definitions of parallel computing terms are found in *"A Glossary of Parallel Computing Terminology"* [5]

is called a **speedup**.

A linear speedup — *a speedup directly proportional to the number of processors used — corresponds to the maximum efficiency of a parallel algorithm.*

Two major laws govern the speedup of a parallel implementation.

Theorem 2 Amdahl's Law

If σ is the proportion of a calculation which is serial, and $1-\sigma$ is the parallelizable portion, then the speedup which can be achieved with P processors is

$$S = \lim_{P\to\infty} \frac{1}{\sigma - \frac{1-\sigma}{P}} = \frac{1}{\sigma}.$$

Thus, no matter how many processors are used, the speedup is limited.

Theorem 3 Gustafson's Law

If the size of most problems is scaled up sufficiently, then any required speedup efficiency can be achieved on any number of processors.

In this paper we will use a notion of a speedup as a ratio between τ_s and τ_P, where τ_s is the execution time of a sequential implicit backward finite difference algorithm. This choice can be justified by the fact that this method is applied for all interior points of sub-domains in every studied parallel scheme. Moreover all parallel implementations share the same code for solving linear equations and error estimates used in the sequential implicit backward difference method.

A non-homogeneous equation (1.2) with a right part (1.3) was used for all the performance graphs in this paper. The problem is solved for $N = 1024$ points in the domain with $M = 8192$ time-levels. All numerical experiments were computed using at most 8 processors out of 60 available on a MYRIAS SPS-2 computer. As it will be shown further the performance of the presented algorithm was low even for this number of processors. Moreover the proposed new method of communication reduction limits a possible number of sub-domains to 8 for 1024×8192 problem size.

Chapter 2

Direct Implementation of Domain Decomposition Algorithm

2.1 Algorithm pseudocode, data structures and shared memory model

Let us start a discussion of the direct implementation of the computation scheme proposed in [2] "translating" mathematical description of the method (1.14)-(1.18) into parallel pseudocode. The following data structure **Domain** represents one sub-domain of the problem (Figure 1).

```
typedef struct {
        int max;        /* The numbe. of points in the sub-domain */
        int start;      /* The first point of the sub-domain */

        double *A ;     /* a matrix for implicit backward finite difference
                        method */
        double *F;      /* right part of the heat equation */

        /*
        These shared memory locations will contain points of my solution
        and will be read by my neighbors
        */
        volatile BC *leftInternBC;
        volatile BC *rightInternBC;
```

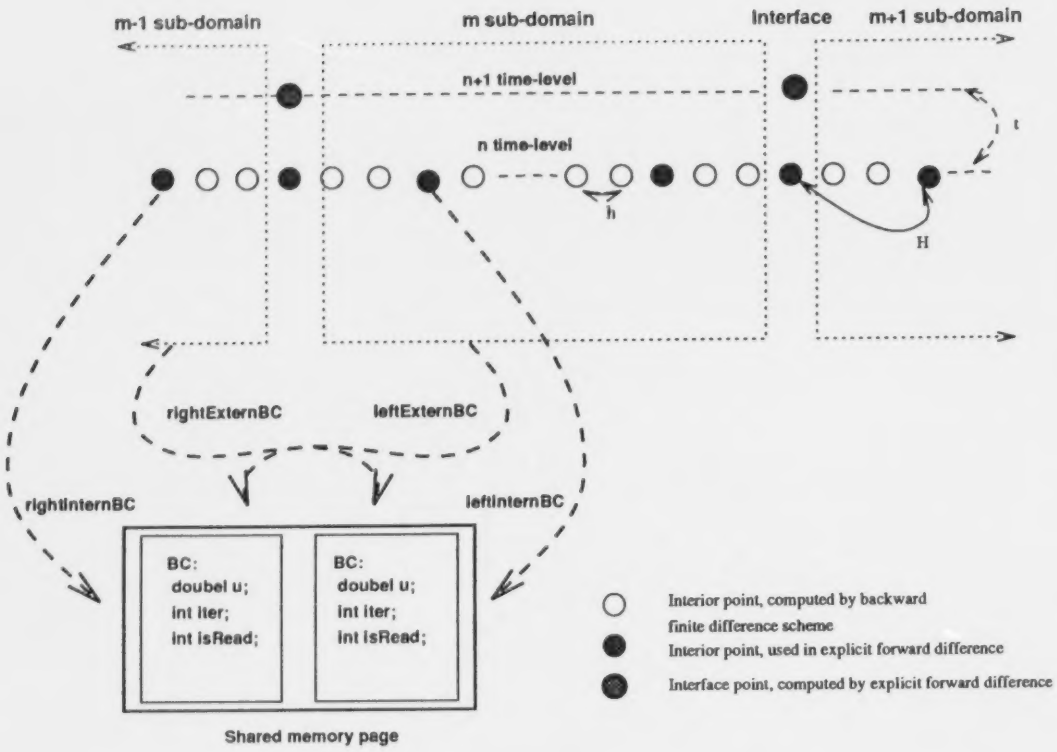

Figure 1: The sub-domain data structures

```
        /*
        These shared memory locations will contain points of my
        neighbor's solution and will be read by me
        */
        volatile BC *leftExternBC;
        volatile BC *rightExternBC;

        } Domain;

        /* The shared memory location */

typedef struct {
        double u;       /* One point of the solution */
        int iter;       /* Iteration when this point was computed */
        int isRead;     /* A flag to mark if the value was read already
                        by my neighbor. This information is needed
                        to avoid a deadlock*/
        } BC;
```

A function computeDomain represents an algorithm for computing one of the sub-domains. This code will be executed in parallel by every processor in the computational domain.

Function 2.1: computeDomain

```
void computeDomain(domain)
Domain *domain;
{
double t, left, right, leftBC, rightBC;
int j, il, ir, iter;

Allocate local memory  for the solution on this sub-domain,
perform LU decomposition of the matrix for implicit backward
difference method.

for(iter = 1; iter <= MAXITERATION; i++ )
{
        /*
        Read  my leftExternBC, written by my left neighbor on the previous
        iteration and compute my left boundary condition (left Interface point)
        */

        if ( domain->leftExternBC )
                {
                left = memread( domain->leftExternBC, iter -1,waited);
                U[domain->start] = leftBC = forwardBC (    left,
                                        U[domain->start],
                                        U[domain->start+offBC],
                                        domain->start, iter );
                }
        /*
        Read my rightExternBC, written by my right neighbor on the previous
        iteration and compute my right boundary condition
        (right Interface point)
        */

        if ( domain->rightExternBC )
                {
                right = memread( domain->rightExternBC, iter -1,waited) ;
                domain->rightU = rightBC =
                forwardBC ( U[domain->start+domain->max-offBC],
                                        domain->rightU,
                                        right,
                                        domain->start+domain->max, iter );
                }
        else
                domain->rightU = rightBC = 0.0;
```

```
        Compute my internal points of the sub-domain using
        implicit backward finite difference method

        /*
        Update my rightInternBC and leftInternBC. They will be read
        by my neighbors on the next iteration
        */

        if (! domain.isFirst )
        {
                memwrite (U[domain.start + offBC], iter,
                          domain.leftInternBC, slowed);
        }

        if ( domain.rightInternBC )
        {
                memwrite (U[domain.start + domain.max - offBC], iter,
                          domain.rightInternBC, slowed);
        }
}
}
```

As it is seen from the pseudocode of `computeDomain` all communication between neighboring processors is implemented through the shared memory (functions `memread` and `memwrite`). Moreover, these two functions are responsible for synchronization of the algorithm. Notice that variables `waited` and `slow` keep statistics on the number of iterations when computation must be stopped in order to synchronize the process with neighboring sub-domains and to avoid a deadlock. A counter `waited` contains the number of time-levels when the necessary information is not available; `slow` counts the number of time-levels when a processor must slow down in order to allow its neighbor to read a value from the shared memory before it will be overwritten.

Function 2.2: `memread`

```
double memread( bc, iter, waited)
int     iter;
BC      *bc;
int     *waited;
```

```
{
        Check if the information in the shared memory was already
        updated by the neighboring processor

        isClash=bc -> iter - iter;
        if (isClash)
        {
                If not , then increment a counter "waited"
                and wait till the needed information will be available
        }

        Update the field  "isRead" and return the value, read from the
        shared memory.
        bc -> isRead = iter;
        result = bc->u;
        return(result);

}
```

Function 2.3: memwrite

```
void memwrite( U, iter, bc, slow)
double  U;
int     iter;
BC      *bc;
int     *slow;
{

        Before writing to the shared memory location, make sure that
        the neighbor read the current  value from it.
        if( bc-> isRead != (iter-1))
                {
                Increment counter "slow" and wait till the current
                value in the shared memory is read.
                }

        Write to the shared memory location:
        bc->u = U;
        bc->iter = iter;
}
```

All experiments in this paper were performed on a 64 processor MYRIAS SPS-2 computer. The traditional MYRIAS architecture does not have any shared memory,

though it is simulated by the software. [1]

There are two models of the virtual shared memory, implemented on the MYRIAS: cached and un-cached memory. They can be best characterized by their respective access algorithms.

The cached memory maintains one "write only page" of shared memory, which is available to only one processor at a time. There are multiple "read only " copies of the "writable page" attached to each processor. The user is responsible for broadcasting changes of the "write only" page to its copies. The un-cached memory consists of only one copy of shared memory page, that is floating between processors, and there is no need for broadcasting any changes. This type of memory can be effective for some types problems.

For the domain decomposition method, interfaces between each pair of domains are needed to be stored in the shared memory. Only two neighboring processors will need to access each border of their domains (Figure 1 and Function 2.1). Therefore, there is no need to have multiple copies of the interface points. This leads to the decision to use un-cached memory for this problem.

The graph on Figure 2 represents results of one of the early implementation tests. The heat equation is solved for 256 points in the domain and 1024 time-levels. When the domain is divided into two sub-domains the parallel program will run almost twice faster than the sequential implementation of the backward finite difference method. There is only one interface between two domains and one page of shared memory is moving between the two processors. But as the number of processors increases, the number of borders between domains increases as well. All the data by default is still stored on one page of shared memory, therefore all the processors are competing to access it. The computing time of a parallel program in the example on Figure 2 increases almost exponentially with respect to a number of processors !

The obvious solution is to increase a number of pages of shared memory. The cached memory model is one possible alternative. It will increase a number of "read

[1] See the release notes of MYRIAS system software and the programmers guide [3] and [4]

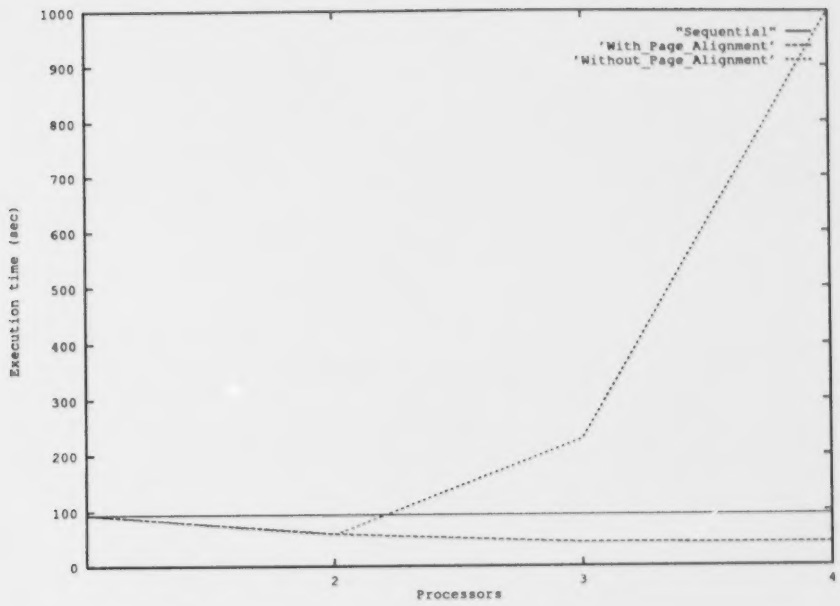

Figure 2: Influence of page alignment of shared memory

only" pages and processors waiting for the update of their interface points are not competing with each other. On the other hand there is only one "writable page" and the update of this page will create a tree of messages to every "read only page", which is not necessary for every processor.

The current implementation uses a page alignment for un-cached model to solve the problem of memory contention.[2] The data needed for a pair of processors is stored on a separate page of un-cached memory. Therefore there are only two processors that will compete for access to a page of shared memory. Improved results are seen from the graph on Figure 2.

[2]The final implementation uses the un-cached memory model, but in the early stages of this project cached memory was tested as well. On average the cached memory version of the program was up to 25% slower then the un-cached one.

2.2 Dependance of the speedup on a problem size

From the description of the explicit/implicit domain decomposition method (1.14)-(1.18) it is clear that all the computations can be done in parallel. Therefore, by Amdahl's Law there is no upper bound on the reachable speedup. Moreover, the Gustafson's Law states that arbitrary high speedup can be reached for the sufficiently scaled problem size.

In the first experiment let us compare the speedup and accuracy of approximation of the domain decomposition algorithm for different problem sizes. We can fix a number of time-levels to $M = 8192$ and change the number of points in the domain $N = 256, 512, 1024, 8192$. A proportion between the number of time-levels and a domain size can be characterized by $s = \tau/h^2$. A sequential implementation of the implicit backward finite difference method will be compared with the parallel domain decomposition ran on 4 processors.

The Table 2.1 compares performance of the sequential and parallel programs. More detailed statistics can be found in the Appendix (Table A.9).

Table 2.1: Dependance of the speedup and approximation errors on a problem size

Problem size			Sequential		Parallel (4 proc)		
M	N	s	Max error	τ_s	Max error	τ_p	τ_s/τ_p
8192	256	8	7.0×10^{-06}	657	6.8×10^{-06}	415	1.58
8192	512	32	5.3×10^{-06}	1314	4.6×10^{-06}	572	2.29
8192	1024	128	4.9×10^{-06}	2630	4.0×10^{-06}	935	2.81
8192	2048	512	4.8×10^{-06}	5260	3.8×10^{-06}	1606	3.27

Speedups achieved in this test are directly proportional to the domain size of the problem. It is clear, that increasing a number of points in the domain will increase the size of sub-domains and a granularity of the parallel computations, therefore will lead to a better speedups.

On the other hand, the only practically reasonable notion of the problem scale is a reduction of computational errors due to the increased domain size. As it was stated earlier the sequential and parallel algorithms have the same order of approximations:

$O(h^2 + \tau)$. But from the Table 2.1 one can see that for very big domains ($s \gg 1$) the maximum computational error does not decrease significantly. Moreover, in order to keep values of s small we will need to increase the number of time-levels polynomially faster than the number of nodes in the domain. Every time-level represents a synchronization barrier and a communication point of the algorithm, hence communication and synchronization efficiency losses are going to grow polynomially fast in relation to the problem scale.

Conclusion 1 *The above considerations suggest that Gustafson's Law is not directly applicable for the studied algorithm and linear speedups are not theoretically reachable for any scaled problem size without loss of computational accuracy.*

2.3 A geometry of domain decomposition.

One of the ways to address a problem of synchronization is by load balancing, evenly distributing the work amongst available processors. For this particular algorithm a computational complexity is linearly dependent on a sub-domain size. Hence, intuitively, one would like to partition a domain into equal size sub-domains to ensure approximately equal work load for every processor. Efficeincy losses can be evaluated using statistics collected by functions `memread` (Function 2.2) and `memwrite` (Function 2.3):

- The number of times computation was stalled, waiting for the neighboring domain to finish its portion of the work and to supply the data (`waited`).
- The number of times a processor was not able to write to a shared memory location as its neighbor still did not read the previous value from this location (`slow`).

In this section all the experiments were conducted for a problem with $M = 8192$ time-levels and $N = 1024$ points in the domain. We will consider data for 8 processors though results are similar for any other number of CPUs.

Table 2.2: Synchronization overhead of equally partitioned domains

Domain No.	Sub-domain Size	`waited`	`slow`
0	128	8085	168
1	128	4621	165
2	128	7622	165
3	128	2721	167
4	128	6989	180
5	128	230	4
6	128	7087	16
7	128	8111	332

The Table 2.2 presents statistics for the equal domain distribution. The main conclusion from this experiment is the fact that the first and last domains were waiting almost on every iteration. Clearly, computations of these sub-domains utilize boundary conditions of the heat conduction problem and therefore need half the communication, than the interior sub-domains. An average cost of reading a shared memory location is about 200–300 floating point operations.[3] This is equivalent to the CPU time needed to compute values of an additional 30-40 points in every domain. This consideration gave an idea to use so-called "heavy border " work distribution.

The "heavy border" method partitions N nodes of the problem in such a way that the first and the last sub-domains will get about 35 extra nodes than interior sub-domains. As it is seen from Table 2.3 values of `waited` and `slow` are close almost for all processors.

Examining Tables 2.2-2.3 it becomes clear that interior domains of equal sizes are not computed with the same speed. One of the reasons for this is memory contention. It takes approximately the same amount of time for each processor to compute all the points of the interior sub-domain, hence two neigbouring processors will try to read from the same page of shared memory at the same time. A method of "mixed distributions" alternates "big" and "small" sub-domain sizes for CPUs with ID num-

[3]This result is experimental. Exact numbers are not available.

Table 2.3: Synchronization overhead of a "heavy border" distribution

Domain No.	Sub-domain Size	waited	slow
0	154	6759	5
1	119	7327	2
2	119	7963	5
3	119	7000	11
4	119	8111	9
5	119	7394	20
6	119	8146	19
7	154	133	12

bers $1 \ldots p-1$, while assigning the biggest number of points to the processors 0 and p. Statistics for "mixed distribution" method is represented in Table 2.4

Table 2.4: Synchronization overhead of a "mixed distribution"

Domain No.	Sub-domain Size	waited	slow
0	148	8069	20
1	131	4571	17
2	111	8594	34
3	131	872	17
4	111	8790	315
5	131	1117	74
6	113	8422	127
7	148	3036	67

The speedup graph on Figure 3 summarizes all the experiments with direct implementation of a domain decomposition algorithm. A detailed statistics can be found in the Appendix (Tables A.10-A.12).

Conclusion 2 *The following statements follow from the results described above.*

- *Optimal efficiency is not reachable mainly due to the high communication and synchronization cost.*

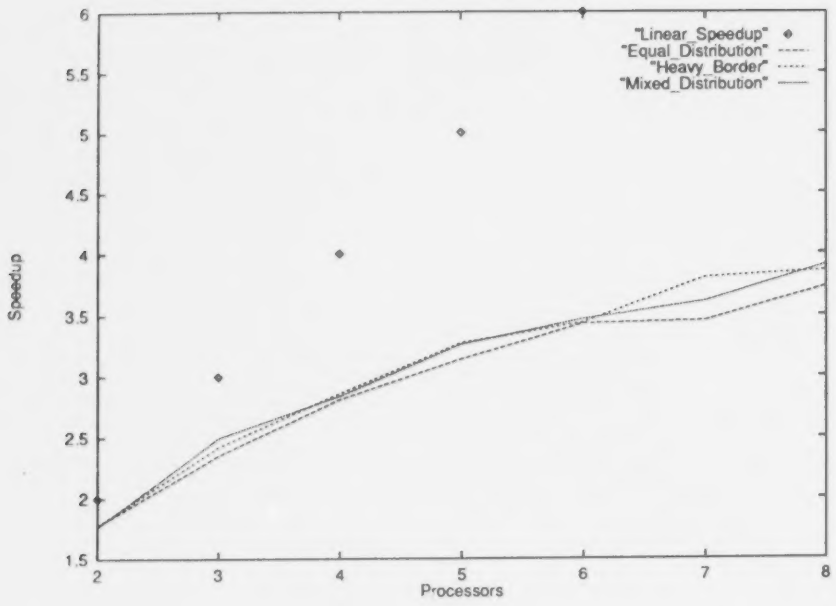

Figure 3: Speedup graph of direct implementations

- *Direct implementation of the proposed computation scheme on a MYRIAS parallel multiprocessor with virtual shared memory gives acceptable speedup only for 2 or 3 processors.*
- *Attempts to improve algorithm performance using different domain sizes can somewhat reduce synchronization overhead and memory contention, but do not give any significant performance gains.*

At the end of this chapter I would like to point out that though a virtual shared memory becomes a bottleneck for this implementation, all addressed problems are not specific to the current MYRIAS architecture. Other implementations on different computers may perform significantly better. Nevertheless all, the outlined problems will just be scaled down or magnified for any parallel multiprocessor.

Chapter 3

Asynchronous Domain Decomposition Algorithm

3.1 Asynchronous chaotic algorithm

One of the main drawbacks of the original algorithm is the requirement that every processor compute the same time-level of its domain at the same time with its neighbors. Consider the following features of the MYRIAS architecture and conditions of the experiments:

- all the processors are of the same type and have the same amount of local memory (4 MB);
- generally speaking, the communication speed between any two processors on one board is the same;
- all the interior sub-domains are of the same size, therefore it takes the same number of processor cycles to compute one time-level.

The analysis above allows us to draw a conclusion, that there are no "slow" and "fast" processors and the synchronization conflicts appear only due to the state of virtual shared memory and the system in general.

Let's try to relax the synchronization and allow every sub-domain to compute its next time-level with currently available values on the interfaces. We can suppose that at least for very smooth functions with small u_t the difference

$$u_i^{n+k} - u_i^n = \epsilon \tag{3.1}$$

is very small; u_i^n is an expected value of the interface point and u_i^{n+k} is the available value of this interface. Suppose, $|k| \leq K$. A relaxation parameter K controls the asynchronous behavior of the algorithm and represents a variety of time-levels that can be computed at the same time by different processors. This approach will allow sub-domains that were slow on the previous time-level to catch up on the next ones. Functions `memread` (2.2) and `memwrite` (2.3) will be modified as follows:

Function 3.4: `memread`

```
double memread( bc, iter,  waited )
int     iter;
BC      *bc;
int *waited;
{
        If the value in the shared memory (bc) was computed more
        then K  iterations ago then wait for my neighbor to catch up.
        isClash = (abs (bc -> iter - iter ) > K );
        if (isClash)
        {
                *waited +=1;
                Sleep, periodically checking the shared memory location (bc)
                till my neighbor will catch up.
        }
        result = bc->u;
        bc -> isRead = iter;
        return(result);
}
```

Function 3.5: `memwrite`

```
void memwrite( U, iter, bc, slow)
double  U;
int     iter;
BC      *bc;
```

```
int     *slow;
{
        char isClash;

        First make sure that my neighbor read from this shared memory location
        at least K iterations ago.
        isClash = (abs( bc -> isRead - iter + 1 ) > K );
        if (isClash)
        {
                *slow += 1 ;
                Sleep, periodically checking the shared memory location (bc)
                till my neighbor will catch up.
        }

        Write to the shared memory.
        bc->u = U;
        bc->iter = iter;
}
```

Using a method, described above, the actual value of every interface point will depend not only on the input data, but on the speed of the neighboring processors and the current state of a whole system. Such computational schemes might cause different final results for the same input and the same number of processors and sub-domains. Further in the paper we will refer to this algorithm as **"chaotic"**.

Let us test the convergence of the chaotic method. Note, a numerical method is convergent if a reduction of the time-step and space-step will result in a reduction of computational errors. In Table 3.5 one can compare maximum errors produced by the proposed asynchronous technique with previously described sequential method and the original synchronized domain decomposition.

Table 3.5: Convergence of the asynchronous method

M	N	Sequential	Synchronized	Asynchronous	
				$K = 8$	$K = 100$
1024	128	4.7×10^{-05}	2.5×10^{-05}	3.9×10^{-03}	2.8×10^{-02}
16392	1024	2.5×10^{-06}	2.2×10^{-06}	1.9×10^{-04}	1.7×10^{-02}

Conclusion 3 *Chaotic asynchronous method gives unacceptably big errors for a tested smooth function in comparison with original synchronized version of the algorithm*

and does not effectively converge to the exact solution. This makes such an approach completely unusable.

3.2 Adaptive asynchronous algorithm

Let us move further from the assumption (3.1) in the previous section and, instead of accepting any available values for the interface point, use an adaptive approach described below.

First, we can rewrite $\frac{1}{r}(rf_r)_r$ of the original heat equation in the following way:

$$\frac{1}{r}(rf_r)_r = \frac{f_r}{r} + f_{rr}. \tag{3.2}$$

For a function $f(r,t)$ define the following approximations [1]

- Right-hand sided formulas:

$$\partial_h f(r) = \frac{11f(r) - 18f(r-h) + 9f(r-2h) - 2f(r-3h)}{6h} \tag{3.3}$$

$$\partial^2{}_h f(r) = \frac{2f(r) - 5f(r-h) + 4f(r-2h) - f(r-3h)}{h^2} \tag{3.4}$$

- Left-hand sided formulas:

$$\partial_h f(r) = \frac{-11f(r) + 18f(r+h) - 9f(r+2h) + 2f(r+3h)}{6h} \tag{3.5}$$

$$\partial^2{}_h f(r) = \frac{2f(r) - 5f(r+h) + 4f(r+2h) - f(r+3h)}{h^2} \tag{3.6}$$

Consider r_i be an interface point between m and $m+1$ sub-domains. Suppose that the m-th sub-domain has completed computations of a time-level n. In order to proceed to $n+1$ time-level, the sub-domain m needs to compute a value of its

[1] Coefficients in these formulas are computed using traditional method of truncated Taylor expansions. We will call (3.3)-(3.6) one-hand sided formulas as they approximate derivative at a point r using values to the right or left from it.

interface point $u(r_i, \tau(n+1))$, using a value of $u(r_i + H, \tau n)$. provided by the sub-domain $m+1$. Assume that sub-domain $m+1$ still has not finished processing $n-th$ time-level and therefore point $u(r_i + H, \tau n)$ is not available.

Under the traditional scheme the m-th processor must stop and wait for the processor $m+1$ to complete the n-th time-level. A chaotic asynchronous method proposes to use a previous time-level value $u(r_i + H, \tau(n-1))$ instead, assuming it to be a sufficiently "close" to the needed value. Alternatively, the m-th sub-domain can adapt to the current state of shared memory and approximate $u(r_i + H, \tau n)$ by using a modified form of the heat equation:

$$\partial_{t,\tau} U_i^n - \frac{\partial_{\bar{H}} U_i^{n-1}}{r} - \partial_{\bar{H}}^2 U_i^{n-1} = 0 \tag{3.7}$$

where $\partial_h f(r)$ and $\partial^2{}_h f(r)$ are computed using (3.3) and (3.4). $\bar{H}$ is a new space-step for one-hand sided approximations, $\bar{H}$ is a multiple of h and usually is bigger then H.

Such an approach allows a processor m to continue computations further, using an approximated values in a forward difference scheme (1.15). The computation model of a described method for the domain m is presented on Figure 4.

Computations for the "slow" processor $m+1$ can be carried out using the same technique. Suppose, that the processor m is working on the $n+k+1$ time-level, while $m+1$ just completed time-level n. In order to proceed the $(m+1)$-th sub-domain needs the value of $u(r_i - H, \tau n)$, but $u(r_i - H, \tau(n+k))$ is available instead. Equation (3.7) can be used to approximate the needed value. Partial derivatives must be computed using (3.5) and (3.6), τk will be used for approximation of $\partial_t U_i^{n+k}$. The computation diagram is analogous to Figure 4.

The function `computeDomain` (3.6) must be changed:

Function 3.6: `computeDomain`

```
void computeDomain(domain)
```

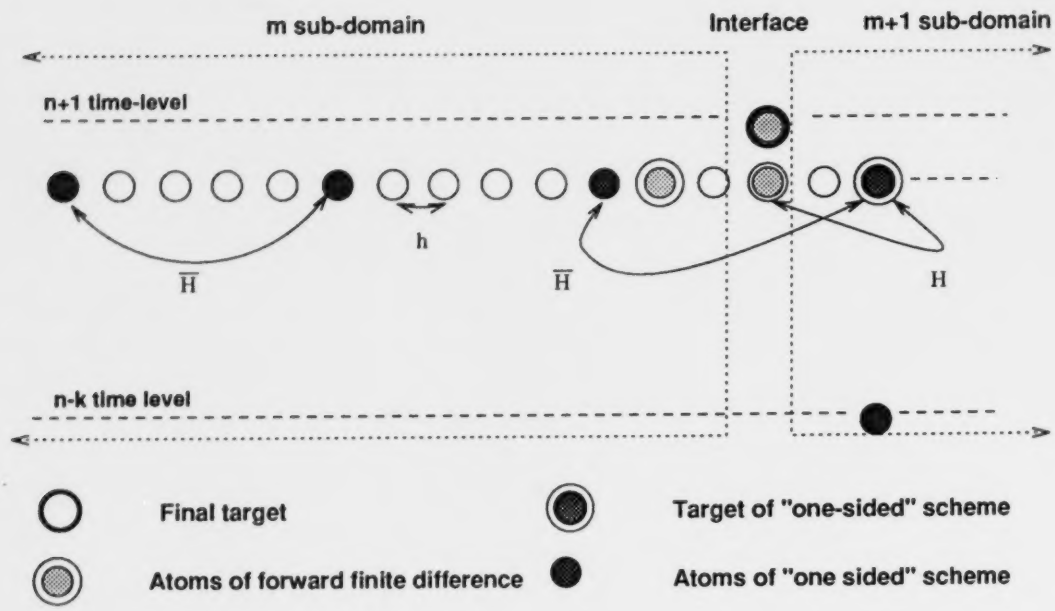

Figure 4: A computation halo of combined correction operator

```
Domain *domain;
{

.....................................

for(iter = 1; iter <= MAXITERATION; i++ )
        {
        if ( domain->leftExternBC )
                {
                Read  my leftExternBC.

                If it was not updated on the previous
                iteration by my left neighbor compute the value of the
                interface point using left-hand sided formula.

                Compute the value of my interface point applying forward
                explicit difference method
                }

        if ( domain->rightExternBC )
                {
                Compute my right interface point in the similar fashion as
                the left interface.
                }
        else
                domain->rightU = rightBC = 0.0;
```

```
            Compute my internal points of the sub-domain using
            implicit backward finite difference method

            Update my rightInternBC and leftInternBC. They will be read
            by my neighbors on the next iteration

            }
}
```

The method, described above, still has a chaotic nature as the actual computation scheme of all interface points will be chosen dynamically, depending on the data in shared memory.

3.3 Stability of asynchronous adaptive algorithm

Let us investigate a *computational stability* of the adaptive asynchronous algorithm. Recall, as it was proven by Theorem 1, the original domain decomposition method is stable under the following condition:

$$\tau \leq \frac{H^2}{2}. \tag{3.8}$$

The main difference between the original and proposed techniques is the change in the computation algorithm for interface points. In the modified version of the method a forward finite difference (1.15) is applied to a point that was previously approximated by one of the one-hand sided formulas (3.3)-(3.6). The inequality (3.8) guarantees stability of the forward finite difference operator. Unfortunately, we cannot expect any one-hand sided formula to be stable by itself. If it was the case, then this will mean that a final solution of a parabolic initial value problem does not depend on one of the boundary conditions. Therefore a proposed chaotic algorithm is stable only if a combination of one-hand sided operator with forward finite difference method is stable.

In order to give a rigorous proof, one has to estimate a norm of the combined approximation operator. In this work we will look at experimental results that suggest that a proposed chaotic scheme is stable at least for a specified class of functions.

A stability requirement for a discussed combined operator can be interpreted in the following way:

Conjecture 1 *The combined operator is stable if the expansion of errors by non-stable one-hand sided operators (3.3)-(3.6) is going to be compensated by the error reduction of conditionally stable forward difference operator.*

By analogy with a condition number for a forward difference method, let us define $\bar{s}$:

$$\bar{s} = \frac{\tau}{\bar{H}^2}$$

where $\bar{H}$ is a space-step for the equation (3.7). The goal of numerical tests was to determine the influence of $\bar{s}$ and relaxation parameters K on computational errors produced by this method.

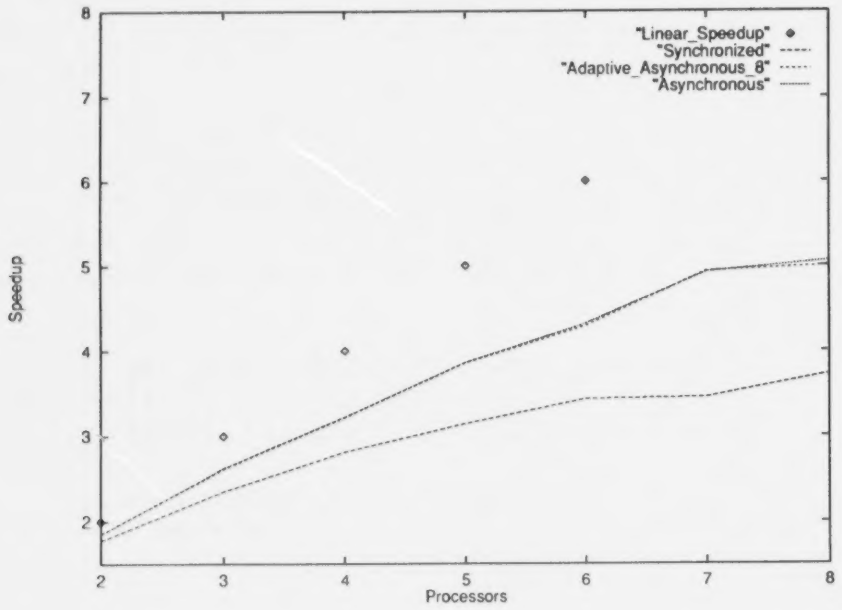

Figure 5: Speedup of a chaotic algorithm with corrections

The Table 3.6 summarizes experimental results for the non-smooth solution[2] (1.6).

[2]Both functions gave similar results.

Table 3.6: Stability of adaptive asynchronous scheme

$\bar{H}/H$	$\bar{s}$	$K=0$	$K=3$	$K=5$	$K=7$
1.5	0.18530	9.1×10^{-04}	∞	...	...
2.5	0.06671	9.1×10^{-04}	9.7×10^{-04}	1.0×10^{-03}	∞
3.0	0.04632	9.1×10^{-04}	9.6×10^{-04}	1.0×10^{-03}	1.0×10^{-03}

The tested domain contained 100 points computed on 1500 time-levels by 3 processors, $\omega = 0.01$. Notice that ∞ in this table corresponds to a non-stable numerical process.

A typical computational log-file is presented by Figure 6. For this experiment $\bar{s} = 0.046$, that corresponds to $\bar{H} = 3H$. The relaxation parameter is $K = 3$. A **gap** in the log specifies a distance between a current time-level and a time-level of a neighbor. One can see that for this set of parameters the input errors are increased by left-hand side backward difference and then reduced by the forward central difference method. Such behavior satisfies a stability condition in Conjunction 1.

Described numerical tests allow us to conclude the following:

Conclusion 4 *The chaotic algorithm with corrections is stable for functions with bounded* $\partial^2 u/\partial t^2$ *for small values of parameter* K *and* $\bar{s} \ll s = \tau/H^2$.

The speedup graph is shown on Figure 5. Performance statistics is presented in Appendix (Tables A.13 and A.14). The process of synchronization reduction tends to reduce the amount of system overhead (compare with `system+waited cpu time` and `pages in/created` in Tables A.10-A.12). Overall proportion of the useful `user cpu time` is increased approximately by 20%.

Figure 6: A typical computation log-file

```
 ........................................
 Iteration 37 is computed

Left backward difference (gap 2)
Input Errors:  2.07e-01 1.76e-01 1.37e-01 2.28e-01
Result   2.295839e-01
Central Difference   error 2.223917e-01

 ........................................
 Iteration 38 is computed

Left backward difference (gap 1)
Input Errors:  2.06e-01 1.75e-01 1.36e-01 2.28e-01
Result   2.287237e-01
Central Difference   error 2.215820e-01
 ........................................

 Iteration 39 is computed
 ........................................
 Iteration 40 is computed

Left backward difference (gap 2)
Input Errors:  2.05e-01 1.74e-01 1.35e-01 2.26e-01
Result   2.273628e-01
Central Difference   error 2.201042e-01
 ........................................
 Iteration 41 is computed
```

Chapter 4

Domain Decomposition with Reduced Communication Scheme

Analyzing performance graphs in the previous chapter one can conclude that, at least for a MYRIAS architecture, the synchronization overhead is not the main reason for low speedups. The original algorithm can be faster if it is possible to reduce the access of shared memory. The presented experimental stability results for a chaotic algorithm with corrections suggests a solution to this problem.

Let us consider the following scheme:

- All the interior points of sub-domains are computed as before.
- Every $K-th$ time-level interface points are computed applying (1.15), where a small number K is a **synchronization parameter**. Only on these time-levels a processor will access shared memory.
- All the time-levels between $K(j)$ and $K(j+1)$ will be computed by a combined operator, previously defined for an adaptive asynchronous scheme.

Such algorithm will increase granularity of computation and reduce the amount of communication by K times.

The new pseudocode of the function `computeDomain` is presented below:

Function 4.7: computeDomain

```
void computeDomain(domain)
Domain *domain;
{

.....................................

for(iter = 1; iter <= MAXITERATION; i++ )

        Read the values from the shared memory, computed by my neighbor
        on the previous iteration, wait if they are not available.

        for(gap = 0; (gap <= K) && (iter < MAXITERATION); iter ++, gap++ )
        {
                Compute the value of my interface points applying
                one-hand sided formulas and forward
                explicit difference method.

                Compute my internal points of the sub-domain using
                implicit backward finite difference method
        }
        Update my rightInternBC and leftInternBC. They will be read
        by my neighbors on the next iteration

}
```

The performance graph on Figure 7 compares all three previously discussed methods. The reduced communication scheme has a better speedup for any number of processors between 2 and 8. Statistics for the reduced communication scheme (Tables A.15-A.17) shows that less frequent access of the shared memory results in the further increase of the `user cpu time` and significant reduction of the number of `created memory pages`.

Table 4.7: Stability of the reduced communication scheme

$\bar{H}/H$	s	$K=0$	$K=3$	$K=5$	$K=7$
2	0.104	9.17×10^{-04}	1.4×10^{-04}	∞	...
2.5	0.0667	9.17×10^{-04}	1.3×10^{-04}	1.9×10^{-04}	∞
3.0	0.04632	9.17×10^{-04}	1.3×10^{-04}	1.7×10^{-04}	2.6×10^{-04}

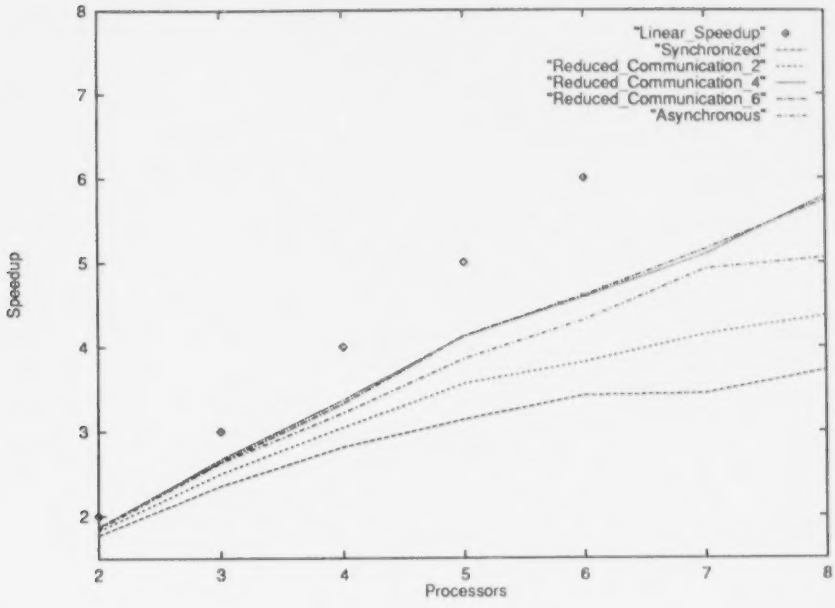

Figure 7: Reduced communication verses other methods

Moreover, a described method can be viewed as a special case of a chaotic method, described before. Hence, a stability conclusion (Conclusion 4 from the previous chapter) can be applied here too.

The same stability tests were performed as before. A problem size is 100×1500. A non-homogeneous part $F(r,t)$ yields an exact solution (1.6) with big second derivative in time near 0. Table 4.7 summarizes the results.

Table 4.8: Dependence of stability on a number of sub-domains.

No. of sub-domains	$K = 3$	$K = 5$	$K = 7$
4	1.2×10^{-05}	2.7×10^{-05}	6.6×10^{-05}
5	2.5×10^{-05}	5.4×10^{-05}	1.7×10^{-04}
6	3.8×10^{-05}	8.0×10^{-05}	∞
7	6.4×10^{-05}	1.0×10^{-04}	3.2×10^{-04}
8	9.0×10^{-05}	1.3×10^{-04}	∞

A comparison of computation errors for the adaptive asynchronous scheme with corrections (Table 3.6) and reduced communication scheme (Table 4.7) shows:

- The stability estimates for both computation methods are the same, as it was predicted before.

- The actual errors for an algorithm with reduced communication are smaller. This can be explained by the fact that an asynchronous scheme applies one-hand sided formulas much more often then a proposed new method.

The other property of a method with reduced communication is that the actual error and stability can depend on a number of used sub-domains . Table 4.8 reproduces results of the experiment with a smooth exact solution (1.4), $\bar{H} = 2.5H$ and a problem size is 1024×8192.

Stability requirement of the discussed method introduces an upper limit to a possible number of sub-domains for a problem of a fixed size. Consider a heat conduction problem with N points in the domain and T time-levels. A "coarse" grid space step H must satisfy a stability condition for the forward difference method and be a multiple of $h = 1/N$. Therefore, the following inequality holds:

$$H = \frac{m}{N} < \sqrt{\frac{2}{T}} \tag{4.1}$$

where m is the number of points of a "fine" grid in one space-step of a "coarse" grid.

It is important to guarantee that the halo (Figure 4) of a combined operator belongs entirely to one sub-domain, otherwise it will extremely complicate a communication scheme. Suppose that a domain is partitioned equally among P processors and the space-step of one-hand sided operator is $\bar{H} = \bar{m}h$. Then the following must be true:

$$3\bar{m} < \frac{N}{P} + m. \tag{4.2}$$

As it is seen from the numerical experiments, in order to ensure stability of a combined operator for a reduction parameter $K = 5$, $\sigma = \bar{H}/H$ must be not less then 2.5.

Consider a problem where $N = 1024$ and $T = 8192$, used for all speedup graphs in this paper. Using formula (4.1) we will get $m = 17$ and for $\sigma = 2.7$, $\bar{m} = 45$. Then from the inequality (4.2) the biggest possible $P = 8$.

Conclusion 5 *Presented results allow us to make the following conclusions:*

- *The communication reduction method allows to achieve much better speedups for the MYRIAS architecture than any other tested algorithm of parallel domain decomposition.*
- *The adaptive combined operator gves stable numerical results for tested functions and* $\bar{H} \geq 2.5H$.
- *The forward difference stability conditions and experimental values of* σ *force to limit a number of sub-domains for any given problem size.*

Conclusion

The numerical results, presented in this paper, summarize a long series of experiments with the domain decomposition technique for a heat equation in cylindrical coordinates. The main goal of this investigation— to reach reasonably high performance of a MYRIAS implementation of the algorithm — was possible to achieve only by introduction of asynchronous schemes of computation and communication reduction. Though for a number of tested functions, the presented algorithm experimentally proved to be stable, a theoretical question about stability of such approach is open. Moreover, an application of such technique can be generalized to parabolic problems in higher dimensions.

Bibliography

[1] Q.Du C. N. Dawson and T.F. Dupont. A finite difference domain decomposition procedure for numerical solution of the heat equation. *Mathematical Computing*, **57**(91)63-71.

[2] Houshi Li, Yanping Lin, Yau Shu Wong. A finite difference domain decomposition for the heat equation in cylindrical coordinates. *Advances in Computer Methods for Partial Differencial Equations*, (92)441-445.

[3] Myrias Computer Corporation. *PAMS 2.5.0 Relese Notes*, June 1990.

[4] Myrias Computer Corporation. *PAMS 4.0.3 Virtual Shared Memory*, September 1992.

[5] Gregory V. Vilson. A glossary of parallel computing terminology. Edinburgh Parallel Computing Center, University of Edinburgh, *IEEE Parallel & Distributed Technology*, (93)I.1

Appendix

This appendix summerizes all the conducted experiments and represents a detailed statistics for each parallel run of domain decomposition algorithm and its modifications. The following data appears in the tables below:

- **user cpu time**–total time spent executing a user code (% of a total execution time).
- **system cpu time**–total time spent executing system code (% of a total execution time).
- **idle cpu time**–total idle time (% of a total execution time). Note, in most cases idle time is equal to 0.
- **wait cpu time**–total wait time (% of a total execution time) A task is resident but unable to execute, waiting for a resource. For all presented algorithms the time spent waiting for synchronization will be added to this category of efficiency losses.
- **pages in/out**–number of pages moved between processors to satisfy task page requests **in**; number of pages moved due to over-commitment of memory on the processors initiating the move **out** (this value is always equal to 0 and does not appear in the tables below).
- **pages created**–number of pages created to support memory merging.

The speedup is calculated as a ratio between parallel execution time and a time spent to solve a problem sequentially by implicit backward finite difference method ($\tau_s = 2630\ sec$).

Table A.9: Performance of the algorithm verses domain size

Domain size (8192 time-levels)	Time distribution (%)				Memory pages		Time	Speedup (4 processors)
	user	system	idle	wait	in	created		
256	44	25	0	31	57	20	415	1.58
512	61	18	0	21	63	25	572	2.29
1024	73	11	0	15	69	30	935	2.81
2048	84	7	0	8	79	38	1606	3.27

Table A.10: Statistics for the equal distribution algorithm

Proc	Time distribution (%)				Memory pages		Time	Speedup
	user	system	idle	wait	in	created		
2	91	4	0	5	42	20	1481	1.77
3	81	8	0	11	53	23	1118	2.35
4	73	11	0	15	69	30	935	2.81
5	66	13	0	21	78	32	837	3.14
6	60	15	0	24	94	38	767	3.42
7	52	16	0	32	108	44	762	3.45
8	50	18	0	32	122	50	706	3.72

Table A.11: Statistics for the "heavy border" distribution algorithm

Proc	Time distribution (%)				Memory pages		Time	Speedup
	user	system	idle	wait	in	created		
2	91	4	0	5	42	20	1479	1.77
3	83	8	0	9	55	25	1085	2.42
4	74	11	0	14	67	28	919	2.86
5	69	13	0	18	78	32	803	3.27
6	60	15	0	24	94	38	765	3.43
7	58	17	0	25	108	44	692	3.80
8	52	18	0	30	117	44	682	3.85

Table A.12: Statistics for the mixed distribution algorithm

Proc	Time distribution (%)				Memory pages		Time	Speedup
	user	system	idle	wait	in	created		
2	91	4	0	5	41	20	1492	1.76
3	86	8	0	6	54	24	1053	2.49
4	74	11	0	14	68	28	926	2.84
5	69	13	0	18	78	32	806	3.26
6	61	15	0	23	94	38	759	3.46
7	55	16	0	28	108	44	729	3.60
8	53	18	0	29	119	47	674	3.90

Table A.13: Statistics for asynchronous algorithm

Proc	Time distribution (%)				Memory pages		Time	Speedup
	user	system	idle	wait	in	created		
2	94	3	1	1	42	20	1420	1.85
3	90	7	1	3	55	25	1001	2.62
4	83	10	3	4	67	28	816	3.22
5	80	12	3	5	78	32	681	3.86
6	75	15	4	6	94	38	609	4.31
7	74	17	2	7	108	44	534	4.92
8	67	17	8	8	117	44	520	5.05

Table A.14: Statistics for adaptive asynchronous algorithm ($K = 8$)

Proc	Time distribution (%)				Memory pages		Time	Speedup
	user	system	idle	wait	in	created		
2	95	4	0	2	42	20	1419	1.85
3	89	7	0	3	55	25	1006	2.61
4	83	11	0	6	67	28	818	3.21
5	80	13	0	6	78	32	682	3.85
6	75	16	0	9	94	38	613	4.29
7	74	18	0	8	108	44	533	4.93
8	67	20	1	13	117	44	526	5

Table A.15: Statistics for the algorithm with reduced communication scheme ($K = 2$)

Proc	Time distribution (%)				Memory pages		Time	Speedup
	user	system	idle	wait	in	created		
2	94	2	0	4	43	22	1438	1.82
3	87	4	0	9	60	27	1052	2.5
4	80	6	0	14	77	36	862	3.05
5	76	7	0	17	88	40	737	3.56
6	68	8	0	23	103	48	688	3.82
7	64	9	1	27	122	56	634	4.14
8	59	9	1	31	137	64	602	4.36

Table A.16: Statistics for the algorithm with reduced communication scheme ($K = 4$)

Proc	Time distribution (%)				Memory pages		Time	Speedup
	user	system	idle	wait	in	created		
2	96	2	0	2	43	22	1406	1.87
3	92	3	0	4	60	27	986	2.66
4	89	5	0	7	77	36	779	3.37
5	87	6	0	7	88	40	638	4.12
6	82	7	0	11	103	48	573	4.58
7	78	8	1	13	122	56	516	5.09
8	78	9	1	12	137	64	454	5.79

Table A.17: Statistics for the algorithm with reduced communication scheme ($K = 6$)

Proc	Time distribution (%)				Memory pages		Time	Speedup
	user	system	idle	wait	in	created		
2	97	1	0	2	43	22	1402	1.87
3	91	3	0	5	60	27	997	2.63
4	88	4	0	8	77	36	789	3.33
5	87	5	0	7	88	40	637	4.12
6	82	6	0	11	103	48	570	4.61
7	79	7	0	13	122	56	510	5.15
8	78	8	1	14	137	64	458	5.74